## 8.3 Mining Sequence Patterns in Transactional Databases

A **sequence database** consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. There are many applications involving sequence data. Typical examples include customer shopping sequences, Web clickstreams, biological sequences, sequences of events in science and engineering, and in natural and social developments. In this section, we study *sequential pattern mining* in transactional databases. In particular, we start with the basic concepts of sequential pattern mining in Section 8.3.1. Section 8.3.2 presents several scalable methods for such mining. Constraint-based sequential pattern mining is described in Section 8.3.3. Periodicity analysis for sequence data is discussed in Section 8.3.4. Specific methods for mining sequence patterns in biological data are addressed in Section 8.4.

### 8.3.1 Sequential Pattern Mining: Concepts and Primitives

"*What is sequential pattern mining?*" **Sequential pattern mining** is the mining of frequently occurring ordered events or subsequences as patterns. An example of a sequential pattern is "*Customers who buy a Canon digital camera are likely to buy an HP color printer within a month.*" For retail data, sequential patterns are useful for shelf placement and promotions. This industry, as well as telecommunications and other businesses, may also use sequential patterns for targeted marketing, customer retention, and many other tasks. Other areas in which sequential patterns can be applied include Web access pattern analysis, weather prediction, production processes, and network intrusion detection. Notice that most studies of sequential pattern mining concentrate on *categorical* (or *symbolic*) *patterns*, whereas numerical curve analysis usually belongs to the scope of trend analysis and forecasting in statistical time-series analysis, as discussed in Section 8.2.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in 1995 [AS95] based on their study of customer purchase sequences, as follows: "*Given a set of sequences, where each sequence consists of a list of events (or elements) and each event consists of a set of items, and given a user-specified minimum support threshold of min_sup, sequential pattern mining finds all* ***frequent*** *subsequences, that is, the subsequences whose occurrence frequency in the set of sequences is no less than min_sup.*"

Let's establish some vocabulary for our discussion of sequential pattern mining. Let $I = \{I_1, I_2, \ldots, I_p\}$ be the set of all *items*. An **itemset** is a nonempty set of items. A **sequence** is an ordered list of **events**. A sequence $s$ is denoted $\langle e_1 e_2 e_3 \cdots e_l \rangle$, where event $e_1$ occurs before $e_2$, which occurs before $e_3$, and so on. Event $e_j$ is also called an **element** of $s$. In the case of customer purchase data, an event refers to a shopping trip in which a customer bought items at a certain store. The event is thus an itemset, that is, an unordered list of items that the customer purchased during the trip. The itemset (or event) is denoted $(x_1 x_2 \cdots x_q)$, where $x_k$ is an item. For brevity, the brackets are omitted if an element has only one item, that is, element $(x)$ is written as $x$. Suppose that a customer made several shopping trips to the store. These ordered events form a sequence for the customer. That is, the customer first bought the items in $s_1$, then later bought the items in $s_2$, and so on. An item can occur at most once in an event of a sequence, but can occur multiple times in different events of a sequence. The number of instances of items in a sequence is called the **length** of the sequence. A sequence with length $l$ is called an $l$-**sequence**. A sequence $\alpha = \langle a_1 a_2 \cdots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \cdots b_m \rangle$, and $\beta$ is a **supersequence** of $\alpha$, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \le j_1 < j_2 < \cdots < j_n \le m$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_n \subseteq b_{j_n}$. For example, if $\alpha = \langle (ab), d \rangle$ and $\beta = \langle (abc), (de) \rangle$, where $a, b, c, d$, and $e$ are items, then $\alpha$ is a subsequence of $\beta$ and $\beta$ is a supersequence of $\alpha$.

A **sequence database**, $S$, is a set of tuples, $\langle SID, s \rangle$, where $SID$ is a *sequence_ID* and $s$ is a sequence. For our example, $S$ contains sequences for all customers of the store. A tuple $\langle SID, s \rangle$ is said to **contain** a sequence $\alpha$, if $\alpha$ is a subsequence of $s$. The **support** of a sequence $\alpha$ in a sequence database $S$ is the number of tuples in the database containing $\alpha$, that is, $support_S(\alpha) = |\, \{\langle SID, s \rangle | (\langle SID, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\} \,|$. It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer *min_sup* as the **minimum support threshold**, a sequence $\alpha$ is **frequent** in sequence database $S$ if $support_S(\alpha) \ge min\_sup$. That is, for sequence $\alpha$ to be frequent, it must occur at least *min_sup* times in $S$. *A frequent sequence is called a* **sequential pattern**. A sequential pattern with length $l$ is called an $l$-**pattern**. The following example illustrates these concepts.

**Example 8.7** **Sequential patterns.** Consider the sequence database, $S$, given in Table 8.1, which will be used in examples throughout this section. Let $min\_sup = 2$. The set of *items* in the database is $\{a, b, c, d, e, f, g\}$. The database contains four sequences.

Let's look at *sequence* 1, which is $\langle a(abc)(ac)d(cf) \rangle$. It has five *events*, namely $(a)$, $(abc)$, $(ac)$, $(d)$, and $(cf)$, which occur in the order listed. Items $a$ and $c$ each appear more than once in different events of the sequence. There are nine instances of items in sequence 1; therefore, it has a *length* of nine and is called a 9-*sequence*. Item $a$ occurs three times in sequence 1 and so contributes three to the length of the sequence. However, the entire sequence contributes only one to the *support* of $\langle a \rangle$. Sequence $\langle a(bc)df \rangle$ is a *subsequence* of sequence 1 since the events of the former are each subsets of events in sequence 1, and the order of events is preserved. Consider subsequence $s = \langle (ab)c \rangle$. Looking at the sequence database, $S$, we see that sequences 1 and 3 are the only ones that *contain* the subsequence $s$. The support of $s$ is thus 2, which satisfies minimum support.

**Table 8.1** A sequence database

| *Sequence_ID* | *Sequence* |
|---|---|
| 1 | $\langle a(abc)(ac)d(cf) \rangle$ |
| 2 | $\langle (ad)c(bc)(ae) \rangle$ |
| 3 | $\langle (ef)(ab)(df)cb \rangle$ |
| 4 | $\langle eg(af)cbc \rangle$ |

Therefore, $s$ is frequent, and so we call it a *sequential pattern*. It is a 3-*pattern* since it is a sequential pattern of length three. ■

This model of sequential pattern mining is an abstraction of customer-shopping sequence analysis. Scalable methods for sequential pattern mining on such data are described in Section 8.3.2, which follows. Many other sequential pattern mining applications may not be covered by this model. For example, when analyzing Web clickstream sequences, gaps between clicks become important if one wants to predict what the next click might be. In DNA sequence analysis, *approximate* patterns become useful since DNA sequences may contain (symbol) insertions, deletions, and mutations. Such diverse requirements can be viewed as *constraint relaxation* or *enforcement*. In Section 8.3.3, we discuss how to extend the basic sequential mining model to *constrained* sequential pattern mining in order to handle these cases.

### 8.3.2 Scalable Methods for Mining Sequential Patterns

Sequential pattern mining is computationally challenging because such mining may generate and/or test a combinatorially explosive number of intermediate subsequences.

"*How can we develop efficient and scalable methods for sequential pattern mining?*" Recent developments have made progress in two directions: (1) efficient methods for mining the *full set* of sequential patterns, and (2) efficient methods for mining only the *set of closed* sequential patterns, where a sequential pattern $s$ is **closed** if there exists no sequential pattern $s'$ where $s'$ is a proper supersequence of $s$, and $s'$ has the same (frequency) support as $s$.[6] Because all of the subsequences of a frequent sequence are also frequent, mining the set of closed sequential patterns may avoid the generation of unnecessary subsequences and thus lead to more compact results as well as more efficient methods than mining the full set. We will first examine methods for mining the full set and then study how they can be extended for mining the closed set. In addition, we discuss modifications for mining multilevel, multidimensional sequential patterns (i.e., with multiple levels of granularity).

The major approaches for mining the full set of sequential patterns are similar to those introduced for frequent itemset mining in Chapter 5. Here, we discuss three such approaches for sequential pattern mining, represented by the algorithms GSP, SPADE, and PrefixSpan, respectively. GSP adopts a *candidate generate-and-test* approach using *horizonal data format* (where the data are represented as $\langle$*sequence_ID* : *sequence_of_itemsets*$\rangle$, as usual, where each itemset is an event). SPADE adopts a candidate generate-and-test approach using *vertical data format* (where the data are represented as $\langle$*itemset* : (*sequence_ID*, *event_ID*)$\rangle$). The vertical data format can be obtained by transforming from a horizontally formatted sequence database in just one scan. PrefixSpan is a *pattern growth* method, which does not require candidate generation.

[6]Closed frequent itemsets were introduced in Chapter 5. Here, the definition is applied to sequential patterns.

All three approaches either directly or indirectly explore the **Apriori property**, stated as follows: *every nonempty subsequence of a sequential pattern is a sequential pattern.* (Recall that for a pattern to be called sequential, it must be frequent. That is, it must satisfy minimum support.) The Apriori property is antimonotonic (or downward-closed) in that, if a sequence cannot pass a test (e.g., regarding minimum support), all of its supersequences will also fail the test. Use of this property to prune the search space can help make the discovery of sequential patterns more efficient.

## GSP: A Sequential Pattern Mining Algorithm Based on Candidate Generate-and-Test

**GSP** (Generalized Sequential Patterns) is a sequential pattern mining method that was developed by Srikant and Agrawal in 1996. It is an extension of their seminal algorithm for frequent itemset mining, known as Apriori (Section 5.2). GSP uses the downward-closure property of sequential patterns and adopts a multiple-pass, candidate generate-and-test approach. The algorithm is outlined as follows. In the first scan of the database, it finds all of the frequent items, that is, those with minimum support. Each such item yields a 1-event frequent sequence consisting of that item. Each subsequent pass starts with a *seed set* of sequential patterns—the set of sequential patterns found in the previous pass. This seed set is used to generate new potentially frequent patterns, called *candidate sequences*. Each candidate sequence contains one more item than the seed sequential pattern from which it was generated (where each event in the pattern may contain one or multiple items). Recall that the number of instances of items in a sequence is the *length* of the sequence. So, all of the candidate sequences in a given pass will have the same length. We refer to a sequence with length $k$ as a $k$-sequence. Let $C_k$ denote the set of candidate $k$-sequences. A pass over the database finds the support for each candidate $k$-sequence. The candidates in $C_k$ with at least *min_sup* form $L_k$, the set of all *frequent* $k$-sequences. This set then becomes the seed set for the next pass, $k+1$. The algorithm terminates when no new sequential pattern is found in a pass, or no candidate sequence can be generated.

The method is illustrated in the following example.

**Example 8.8** **GSP: Candidate generate-and-test (using horizontal data format).** Suppose we are given the same sequence database, $S$, of Table 8.1 from Example 8.7, with *min_sup* = 2. Note that the data are represented in horizontal data format. In the first scan ($k = 1$), GSP collects the support for each item. The set of candidate 1-sequences is thus (shown here in the form of "*sequence:support*"): $\langle a \rangle : 4$, $\langle b \rangle : 4$, $\langle c \rangle : 3$, $\langle d \rangle : 3$, $\langle e \rangle : 3$, $\langle f \rangle : 3$, $\langle g \rangle : 1$.

The sequence $\langle g \rangle$ has a support of only 1 and is the only sequence that does not satisfy minimum support. By filtering it out, we obtain the first seed set, $L_1 = \{\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle e \rangle, \langle f \rangle\}$. Each member in the set represents a 1-event sequential pattern. Each subsequent pass starts with the seed set found in the previous pass and uses it to generate new candidate sequences, which are potentially frequent.

Using $L_1$ as the seed set, this set of six length-1 sequential patterns generates a set of $6 \times 6 + \frac{6 \times 5}{2} = 51$ candidate sequences of length 2, $C_2 = \{\langle aa\rangle, \langle ab\rangle, \ldots, \langle af\rangle, \langle ba\rangle, \langle bb\rangle, \ldots, \langle ff\rangle, \langle (ab)\rangle, \langle (ac)\rangle, \ldots, \langle (ef)\rangle\}$.

In general, the set of candidates is generated by a self-join of the sequential patterns found in the previous pass (see Section 5.2.1 for details). GSP applies the Apriori property to prune the set of candidates as follows. In the $k$-th pass, a sequence is a candidate only if each of its length-$(k-1)$ subsequences is a sequential pattern found at the $(k-1)$-th pass. A new scan of the database collects the support for each candidate sequence and finds a new set of sequential patterns, $L_k$. This set becomes the seed for the next pass. The algorithm terminates when no sequential pattern is found in a pass or when no candidate sequence is generated. Clearly, the number of scans is at least the maximum length of sequential patterns. GSP needs one more scan if the sequential patterns obtained in the last scan still generate new candidates (none of which are found to be frequent).

Although GSP benefits from the Apriori pruning, it still generates a large number of candidates. In this example, six length-1 sequential patterns generate 51 length-2 candidates; 22 length-2 sequential patterns generate 64 length-3 candidates; and so on. Some candidates generated by GSP may not appear in the database at all. In this example, 13 out of 64 length-3 candidates do not appear in the database, resulting in wasted time. ■

The example shows that although an Apriori-like sequential pattern mining method, such as GSP, reduces search space, it typically needs to scan the database multiple times. It will likely generate a huge set of candidate sequences, especially when mining long sequences. There is a need for more efficient mining methods.

### SPADE: An Apriori-Based Vertical Data Format Sequential Pattern Mining Algorithm

The Apriori-like sequential pattern mining approach (based on candidate generate-and-test) can also be explored by mapping a sequence database into vertical data format. In **vertical data format**, the database becomes a set of tuples of the form $\langle itemset : (sequence\_ID, event\_ID)\rangle$. That is, for a given itemset, we record the sequence identifier and corresponding event identifier for which the itemset occurs. The **event identifier** serves as a timestamp within a sequence. The *event_ID* of the $i$th itemset (or event) in a sequence is $i$. Note than an itemset can occur in more than one sequence. The set of (*sequence_ID*, *event_ID*) pairs for a given itemset forms the **ID_list** of the itemset. The mapping from horizontal to vertical format requires one scan of the database. A major advantage of using this format is that we can determine the support of any $k$-sequence by simply joining the ID_lists of any two of its $(k-1)$-length subsequences. The length of the resulting ID_list (i.e., unique *sequence_ID* values) is equal to the support of the $k$-sequence, which tells us whether the sequence is frequent.

**SPADE** (Sequential PAttern Discovery using Equivalent classes) is an Apriori-based sequential pattern mining algorithm that uses vertical data format. As with GSP, SPADE requires one scan to find the frequent 1-sequences. To find candidate 2-sequences, we join all pairs of single items if they are frequent (therein, it applies the Apriori property), if they share the same sequence identifier, and if their event identifiers follow a sequential ordering. That is, the first item in the pair must occur as an event before the second item, where both occur in the same sequence. Similarly, we can grow the length of itemsets from length 2 to length 3, and so on. The procedure stops when no frequent sequences can be found or no such sequences can be formed by such joins. The following example helps illustrate the process.

**Example 8.9** **SPADE: Candidate generate-and-test using vertical data format.** Let *min_sup* = 2. Our running example sequence database, *S*, of Table 8.1 is in horizonal data format. SPADE first scans *S* and transforms it into vertical format, as shown in Figure 8.6(a). Each itemset (or event) is associated with its ID_list, which is the set of *SID* (*sequence_ID*) and *EID* (*event_ID*) pairs that contain the itemset. The ID_list for individual items, *a*, *b*, and so on, is shown in Figure 8.6(b). For example, the ID_list for item *b* consists of the following (*SID*, *EID*) pairs: $\{(1, 2), (2, 3), (3, 2), (3, 5), (4, 5)\}$, where the entry (1, 2) means that *b* occurs in sequence 1, event 2, and so on. Items *a* and *b* are frequent. They can be joined to form the length-2 sequence, $\langle a, b\rangle$. We find the support of this sequence as follows. We join the ID_lists of *a* and *b* by joining on the same *sequence_ID* wherever, according to the *event_ID*s, *a* occurs before *b*. That is, the join must preserve the temporal order of the events involved. The result of such a join for *a* and *b* is shown in the ID_list for *ab* of Figure 8.6(c). For example, the ID_list for 2-sequence *ab* is a set of triples, $(SID, EID(a), EID(b))$, namely $\{(1, 1, 2), (2, 1, 3), (3, 2, 5), (4, 3, 5)\}$. The entry (2, 1, 3), for example, shows that both *a* and *b* occur in sequence 2, and that *a* (event 1 of the sequence) occurs before *b* (event 3), as required. Furthermore, the frequent 2-sequences can be joined (while considering the Apriori pruning heuristic that the ($k$-1)-subsequences of a candidate $k$-sequence must be frequent) to form 3-sequences, as in Figure 8.6(d), and so on. The process terminates when no frequent sequences can be found or no candidate sequences can be formed. Additional details of the method can be found in Zaki [Zak01]. ■

The use of vertical data format, with the creation of ID_lists, reduces scans of the sequence database. The ID_lists carry the information necessary to find the support of candidates. As the length of a frequent sequence increases, the size of its ID_list decreases, resulting in very fast joins. However, the basic search methodology of SPADE and GSP is breadth-first search (e.g., exploring 1-sequences, then 2-sequences, and so on) and Apriori pruning. Despite the pruning, both algorithms have to generate large sets of candidates in breadth-first manner in order to grow longer sequences. Thus, most of the difficulties suffered in the GSP algorithm recur in SPADE as well.

## PrefixSpan: Prefix-Projected Sequential Pattern Growth

*Pattern growth* is a method of frequent-pattern mining that does not require candidate generation. The technique originated in the FP-growth algorithm for transaction databases, presented in Section 5.2.4. The general idea of this approach is as follows: it finds the frequent single items, then compresses this information into a *frequent-pattern*

| *SID* | *EID* | *itemset* |
|---|---|---|
| 1 | 1 | a |
| 1 | 2 | abc |
| 1 | 3 | ac |
| 1 | 4 | d |
| 1 | 5 | cf |
| 2 | 1 | ad |
| 2 | 2 | c |
| 2 | 3 | bc |
| 2 | 4 | ae |
| 3 | 1 | ef |
| 3 | 2 | ab |
| 3 | 3 | df |
| 3 | 4 | c |
| 3 | 5 | b |
| 4 | 1 | e |
| 4 | 2 | g |
| 4 | 3 | af |
| 4 | 4 | c |
| 4 | 5 | b |
| 4 | 6 | c |

(a) vertical format database

| a | | b | | ··· |
|---|---|---|---|---|
| *SID* | *EID* | *SID* | *EID* | ··· |
| 1 | 1 | 1 | 2 | |
| 1 | 2 | 2 | 3 | |
| 1 | 3 | 3 | 2 | |
| 2 | 1 | 3 | 5 | |
| 2 | 4 | 4 | 5 | |
| 3 | 2 | | | |
| 4 | 3 | | | |

(b) ID_lists for some 1-sequences

| ab | | | ba | | | ··· |
|---|---|---|---|---|---|---|
| *SID* | *EID*$(a)$ | *EID*$(b)$ | *SID* | *EID*$(b)$ | *EID*$(a)$ | ··· |
| 1 | 1 | 2 | 1 | 2 | 3 | |
| 2 | 1 | 3 | 2 | 3 | 4 | |
| 3 | 2 | 5 | | | | |
| 4 | 3 | 5 | | | | |

(c) ID_lists for some 2-sequences

| aba | | | | ··· |
|---|---|---|---|---|
| *SID* | *EID*$(a)$ | *EID*$(b)$ | *EID*$(a)$ | ··· |
| 1 | 1 | 2 | 3 | |
| 2 | 1 | 3 | 4 | |

(d) ID_lists for some 3-sequences

**Figure 8.6** The SPADE mining process: (a) vertical format database; (b) to (d) show fragments of the ID_lists for 1-sequences, 2-sequences, and 3-sequences, respectively.

*tree*, or *FP-tree*. The FP-tree is used to generate a set of projected databases, each associated with one frequent item. Each of these databases is mined separately. The algorithm builds prefix patterns, which it concatenates with suffix patterns to find frequent patterns, avoiding candidate generation. Here, we look at **PrefixSpan**, which extends the pattern-growth approach to instead mine sequential patterns.

Suppose that all the items within an event are listed alphabetically. For example, instead of listing the items in an event as, say, $(bac)$, we list them as $(abc)$ without loss of generality. Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n \rangle$ (where each $e_i$ corresponds to a frequent event in a sequence database, $S$), a sequence $\beta = \langle e'_1 e'_2 \cdots e'_m \rangle$ $(m \leq n)$ is called a **prefix** of $\alpha$ if and only if (1) $e'_i = e_i$ for $(i \leq m-1)$; (2) $e'_m \subseteq e_m$; and (3) all the frequent items in $(e_m - e'_m)$ are alphabetically after those in $e'_m$. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n \rangle$ is called the **suffix** of $\alpha$ with respect to prefix $\beta$, denoted as $\gamma = \alpha/\beta$, where $e''_m = (e_m - e'_m)$.[7] We also denote $\alpha = \beta \cdot \gamma$. Note if $\beta$ is not a subsequence of $\alpha$, the suffix of $\alpha$ with respect to $\beta$ is empty.

We illustrate these concepts with the following example.

**Example 8.10** **Prefix and suffix.** Let sequence $s = \langle a(abc)(ac)d(cf)\rangle$, which corresponds to sequence 1 of our running example sequence database. $\langle a\rangle$, $\langle aa\rangle$, $\langle a(ab)\rangle$, and $\langle a(abc)\rangle$ are four prefixes of $s$. $\langle (abc)(ac)d(cf)\rangle$ is the suffix of $s$ with respect to the prefix $\langle a\rangle$; $\langle (\_bc)(ac)d(cf)\rangle$ is its suffix with respect to the prefix $\langle aa\rangle$; and $\langle (\_c)(ac)d(cf)\rangle$ is its suffix with respect to the prefix $\langle a(ab)\rangle$. ■

Based on the concepts of prefix and suffix, the problem of mining sequential patterns can be decomposed into a set of subproblems as shown:

1. Let $\{\langle x_1\rangle, \langle x_2\rangle, \ldots, \langle x_n\rangle\}$ be the complete set of length-1 sequential patterns in a sequence database, $S$. The complete set of sequential patterns in $S$ can be partitioned into $n$ disjoint subsets. The $i^{th}$ subset $(1 \le i \le n)$ is the set of sequential patterns with prefix $\langle x_i\rangle$.
2. Let $\alpha$ be a length-$l$ sequential pattern and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all length-$(l+1)$ sequential patterns with prefix $\alpha$. The complete set of sequential patterns with prefix $\alpha$, except for $\alpha$ itself, can be partitioned into $m$ disjoint subsets. The $j^{th}$ subset $(1 \le j \le m)$ is the set of sequential patterns prefixed with $\beta_j$.

Based on this observation, the problem can be partitioned recursively. That is, each subset of sequential patterns can be further partitioned when necessary. This forms a *divide-and-conquer* framework. To mine the subsets of sequential patterns, we construct corresponding *projected databases* and mine each one recursively.

Let's use our running example to examine how to use the prefix-based projection approach for mining sequential patterns.

**Example 8.11** **PrefixSpan: A pattern-growth approach.** Using the same sequence database, $S$, of Table 8.1 with $min\_sup = 2$, sequential patterns in $S$ can be mined by a prefix-projection method in the following steps.

1. *Find length-1 sequential patterns.* Scan $S$ once to find all of the frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are $\langle a\rangle : 4$, $\langle b\rangle : 4$, $\langle c\rangle : 4$, $\langle d\rangle : 3$, $\langle e\rangle : 3$, and $\langle f\rangle : 3$, where the notation "$\langle pattern\rangle : count$" represents the pattern and its associated support count.

[7] If $e''_m$ is not empty, the suffix is also denoted as $\langle (\_ \text{ items in } e''_m) e_{m+1} \cdots e_n\rangle$.

**Table 8.2** Projected databases and sequential patterns

| *prefix* | *projected database* | *sequential patterns* |
|---|---|---|
| $\langle a\rangle$ | $\langle(abc)(ac)d(cf)\rangle$, $\langle(\_d)c(bc)(ae)\rangle$, $\langle(\_b)(df)eb\rangle$, $\langle(\_f)cbc\rangle$ | $\langle a\rangle$, $\langle aa\rangle$, $\langle ab\rangle$, $\langle a(bc)\rangle$, $\langle a(bc)a\rangle$, $\langle aba\rangle$, $\langle abc\rangle$, $\langle(ab)\rangle$, $\langle(ab)c\rangle$, $\langle(ab)d\rangle$, $\langle(ab)f\rangle$, $\langle(ab)dc\rangle$, $\langle ac\rangle$, $\langle aca\rangle$, $\langle acb\rangle$, $\langle acc\rangle$, $\langle ad\rangle$, $\langle adc\rangle$, $\langle af\rangle$ |
| $\langle b\rangle$ | $\langle(\_c)(ac)d(cf)\rangle$, $\langle(\_c)(ae)\rangle$, $\langle(df)cb\rangle$, $\langle c\rangle$ | $\langle b\rangle$, $\langle ba\rangle$, $\langle bc\rangle$, $\langle(bc)\rangle$, $\langle(bc)a\rangle$, $\langle bd\rangle$, $\langle bdc\rangle$, $\langle bf\rangle$ |
| $\langle c\rangle$ | $\langle(ac)d(cf)\rangle$, $\langle(bc)(ae)\rangle$, $\langle b\rangle$, $\langle bc\rangle$ | $\langle c\rangle$, $\langle ca\rangle$, $\langle cb\rangle$, $\langle cc\rangle$ |
| $\langle d\rangle$ | $\langle(cf)\rangle$, $\langle c(bc)(ae)\rangle$, $\langle(\_f)cb\rangle$ | $\langle d\rangle$, $\langle db\rangle$, $\langle dc\rangle$, $\langle dcb\rangle$ |
| $\langle e\rangle$ | $\langle(\_f)(ab)(df)cb\rangle$, $\langle(af)cbc\rangle$ | $\langle e\rangle$, $\langle ea\rangle$, $\langle eab\rangle$, $\langle eac\rangle$, $\langle eacb\rangle$, $\langle eb\rangle$, $\langle ebc\rangle$, $\langle ec\rangle$, $\langle ecb\rangle$, $\langle ef\rangle$, $\langle efb\rangle$, $\langle efc\rangle$, $\langle efcb\rangle$. |
| $\langle f\rangle$ | $\langle(ab)(df)cb\rangle$, $\langle cbc\rangle$ | $\langle f\rangle$, $\langle fb\rangle$, $\langle fbc\rangle$, $\langle fc\rangle$, $\langle fcb\rangle$ |

2. *Partition the search space.* The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones with prefix $\langle a\rangle$, (2) the ones with prefix $\langle b\rangle$, ..., and (6) the ones with prefix $\langle f\rangle$.

3. *Find subsets of sequential patterns.* The subsets of sequential patterns mentioned in step 2 can be mined by constructing corresponding *projected databases* and mining each recursively. The projected databases, as well as the sequential patterns found in them, are listed in Table 8.2, while the mining process is explained as follows:

   (a) *Find sequential patterns with prefix $\langle a\rangle$.* Only the sequences containing $\langle a\rangle$ should be collected. Moreover, in a sequence containing $\langle a\rangle$, only the subsequence prefixed with the first occurrence of $\langle a\rangle$ should be considered. For example, in sequence $\langle(ef)(ab)(df)cb\rangle$, only the subsequence $\langle(\_b)(df)cb\rangle$ should be considered for mining sequential patterns prefixed with $\langle a\rangle$. Notice that $(\_b)$ means that the last event in the prefix, which is $a$, together with $b$, form one event.

   The sequences in $S$ containing $\langle a\rangle$ are projected with respect to $\langle a\rangle$ to form the *$\langle a\rangle$-projected database*, which consists of four suffix sequences: $\langle(abc)(ac)d(cf)\rangle$, $\langle(\_d)c(bc)(ae)\rangle$, $\langle(\_b)(df)cb\rangle$, and $\langle(\_f)cbc\rangle$.

   By scanning the $\langle a\rangle$-projected database once, its locally frequent items are identified as $a:2$, $b:4$, $\_b:2$, $c:4$, $d:2$, and $f:2$. Thus all the length-2 sequential patterns prefixed with $\langle a\rangle$ are found, and they are: $\langle aa\rangle:2$, $\langle ab\rangle:4$, $\langle(ab)\rangle:2$, $\langle ac\rangle:4$, $\langle ad\rangle:2$, and $\langle af\rangle:2$.

Recursively, all sequential patterns with prefix $\langle a\rangle$ can be partitioned into six subsets: (1) those prefixed with $\langle aa\rangle$, (2) those with $\langle ab\rangle, \ldots$, and finally, (6) those with $\langle af\rangle$. These subsets can be mined by constructing respective projected databases and mining each recursively as follows:

i. The $\langle aa\rangle$-projected database consists of two nonempty (suffix) subsequences prefixed with $\langle aa\rangle$: $\{\langle(\_bc)(ac)d(cf)\rangle, \{\langle(\_e)\rangle\}$. Because there is no hope of generating any frequent subsequence from this projected database, the processing of the $\langle aa\rangle$-projected database terminates.

ii. The $\langle ab\rangle$-projected database consists of three suffix sequences: $\langle(\_c)(ac)d(cf)\rangle$, $\langle(\_c)a\rangle$, and $\langle c\rangle$. Recursively mining the $\langle ab\rangle$-projected database returns four sequential patterns: $\langle(\_c)\rangle$, $\langle(\_c)a\rangle$, $\langle a\rangle$, and $\langle c\rangle$ (i.e., $\langle a(bc)\rangle$, $\langle a(bc)a\rangle$, $\langle aba\rangle$, and $\langle abc\rangle$.) They form the complete set of sequential patterns prefixed with $\langle ab\rangle$.

iii. The $\langle(ab)\rangle$-projected database contains only two sequences: $\langle(\_c)(ac)\,d(cf)\rangle$ and $\langle(df)cb\rangle$, which leads to the finding of the following sequential patterns prefixed with $\langle(ab)\rangle$: $\langle c\rangle$, $\langle d\rangle$, $\langle f\rangle$, and $\langle dc\rangle$.

iv. The $\langle ac\rangle$-, $\langle ad\rangle$-, and $\langle af\rangle$- projected databases can be constructed and recursively mined in a similar manner. The sequential patterns found are shown in Table 8.2.

(b) *Find sequential patterns with prefix $\langle b\rangle$, $\langle c\rangle$, $\langle d\rangle$, $\langle e\rangle$, and $\langle f\rangle$, respectively*. This can be done by constructing the $\langle b\rangle$-, $\langle c\rangle$-, $\langle d\rangle$-, $\langle e\rangle$-, and $\langle f\rangle$-projected databases and mining them respectively. The projected databases as well as the sequential patterns found are also shown in Table 8.2.

**4.** *The set of sequential patterns is the collection of patterns found in the above recursive mining process.* ■

The method described above generates no candidate sequences in the mining process. However, it may generate many projected databases, one for each frequent prefix-subsequence. Forming a large number of projected databases recursively may become the major cost of the method, if such databases have to be generated physically. An important optimization technique is **pseudo-projection**, which registers the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence instead of performing physical projection. That is, a physical projection of a sequence is replaced by registering a sequence identifier and the projected position index point. Pseudo-projection reduces the cost of projection substantially when such projection can be done in main memory. However, it may not be efficient if the pseudo-projection is used for disk-based accessing because random access of disk space is costly. The suggested approach is that if the original sequence database or the projected databases are too big to fit in memory, the physical projection should be applied; however, the execution should be swapped to pseudo-projection once the projected databases can fit in memory. This methodology is adopted in the PrefixSpan implementation.

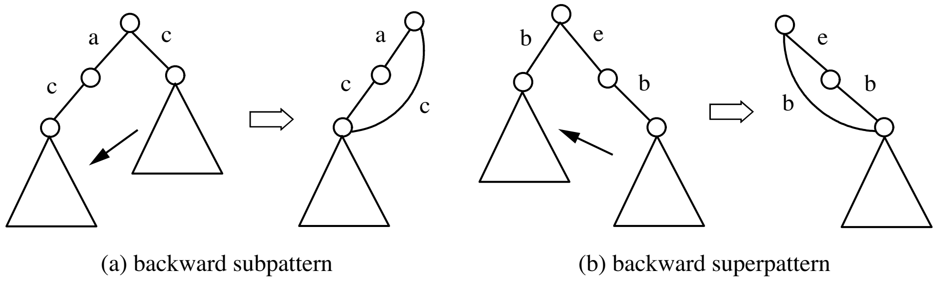

**Figure 8.7** A backward subpattern and a backward superpattern.

A performance comparison of GSP, SPADE, and PrefixSpan shows that PrefixSpan has the best overall performance. SPADE, although weaker than PrefixSpan in most cases, outperforms GSP. Generating huge candidate sets may consume a tremendous amount of memory, thereby causing candidate generate-and-test algorithms to become very slow. The comparison also found that when there is a large number of frequent subsequences, all three algorithms run slowly. This problem can be partially solved by closed sequential pattern mining.

## Mining Closed Sequential Patterns

Because mining the complete set of frequent subsequences can generate a huge number of sequential patterns, an interesting alternative is to mine frequent *closed subsequences* only, that is, those containing no supersequence with the same support. Mining closed sequential patterns can produce a significantly less number of sequences than the full set of sequential patterns. Note that the full set of frequent subsequences, together with their supports, can easily be derived from the closed subsequences. Thus, closed subsequences have the same expressive power as the corresponding full set of subsequences. Because of their compactness, they may also be quicker to find.

**CloSpan** is an efficient closed sequential pattern mining method. The method is based on a property of sequence databases, called **equivalence of projected databases**, stated as follows: *Two projected sequence databases, $S|_\alpha = S|_\beta$,*[8] *$\alpha \sqsubseteq \beta$ (i.e., $\alpha$ is a subsequence of $\beta$), are equivalent if and only if the total number of items in $S|_\alpha$ is equal to the total number of items in $S|_\beta$.*

Based on this property, CloSpan can prune the nonclosed sequences from further consideration during the mining process. That is, whenever we find two prefix-based projected databases that are exactly the same, we can stop growing one of them. This can be used to prune *backward subpatterns* and *backward superpatterns* as indicated in Figure 8.7.

[8]In $S|_\alpha$, a sequence database $S$ is projected with respect to sequence (e.g., prefix) $\alpha$. The notation $S|_\beta$ can be similarly defined.

After such pruning and mining, a postprocessing step is still required in order to delete nonclosed sequential patterns that may exist in the derived set. A later algorithm called BIDE (which performs a bidirectional search) can further optimize this process to avoid such additional checking.

Empirical results show that CloSpan often derives a much smaller set of sequential patterns in a shorter time than PrefixSpan, which mines the complete set of sequential patterns.

### Mining Multidimensional, Multilevel Sequential Patterns

Sequence identifiers (representing individual customers, for example) and sequence items (such as products bought) are often associated with additional pieces of information. Sequential pattern mining should take advantage of such additional information to discover interesting patterns in multidimensional, multilevel information space. Take customer shopping transactions, for instance. In a sequence database for such data, the additional information associated with sequence IDs could include customer age, address, group, and profession. Information associated with items could include item category, brand, model type, model number, place manufactured, and manufacture date. Mining *multidimensional, multilevel* sequential patterns is the discovery of interesting patterns in such a broad dimensional space, at different levels of detail.

**Example 8.12 Multidimensional, multilevel sequential patterns.** The discovery that "*Retired customers who purchase a digital camera are likely to purchase a color printer within a month*" and that "*Young adults who purchase a laptop are likely to buy a flash drive within two weeks*" are examples of multidimensional, multilevel sequential patterns. By grouping customers into "*retired customers*" and "*young adults*" according to the values in the age dimension, and by generalizing items to, say, "*digital camera*" rather than a specific model, the patterns mined here are associated with additional dimensions and are at a higher level of granularity. ■

"*Can a typical sequential pattern algorithm such as PrefixSpan be extended to efficiently mine multidimensional, multilevel sequential patterns?*" One suggested modification is to associate the multidimensional, multilevel information with the *sequence_ID* and *item_ID*, respectively, which the mining method can take into consideration when finding frequent subsequences. For example, (*Chicago*, *middle_aged*, *business*) can be associated with *sequence_ID_1002* (for a given customer), whereas (*Digital_camera*, *Canon*, *Supershot*, *SD*400, *Japan*, 2005) can be associated with *item_ID_543005* in the sequence. A sequential pattern mining algorithm will use such information in the mining process to find sequential patterns associated with multidimensional, multilevel information.

## 8.3.3 Constraint-Based Mining of Sequential Patterns

As shown in our study of frequent-pattern mining in Chapter 5, mining that is performed without user- or expert-specified constraints may generate numerous patterns that are of no interest. Such unfocused mining can reduce both the efficiency and usability of frequent-pattern mining. Thus, we promote **constraint-based mining**, which incorporates user-specified constraints to reduce the search space and derive only patterns that are of interest to the user.

Constraints can be expressed in many forms. They may specify desired relationships between attributes, attribute values, or aggregates within the resulting patterns mined. Regular expressions can also be used as constraints in the form of "pattern templates," which specify the desired form of the patterns to be mined. The general concepts introduced for constraint-based frequent pattern mining in Section 5.5.1 apply to constraint-based sequential pattern mining as well. The key idea to note is that these kinds of constraints can be used *during* the mining process to confine the search space, thereby improving (1) the efficiency of the mining and (2) the interestingness of the resulting patterns found. This idea is also referred to as *"pushing the constraints deep into the mining process."*

We now examine some typical examples of constraints for sequential pattern mining. First, constraints can be related to the **duration**, $T$, of a sequence. The duration may be the maximal or minimal length of the sequence in the database, or a user-specified duration related to time, such as the year 2005. Sequential pattern mining can then be confined to the data within the specified duration, $T$.

Constraints relating to the maximal or minimal length (duration) can be treated as *antimonotonic* or *monotonic* constraints, respectively. For example, the constraint $T \leq 10$ is **antimonotonic** since, if a sequence does not satisfy this constraint, then neither will any of its supersequences (which are, obviously, longer). The constraint $T > 10$ is **monotonic**. This means that if a sequence satisfies the constraint, then all of its supersequences will also satisfy the constraint. We have already seen several examples in this chapter of how antimonotonic constraints (such as those involving minimum support) can be pushed deep into the mining process to prune the search space. Monotonic constraints can be used in a way similar to its frequent-pattern counterpart as well.

Constraints related to a specific duration, such as a particular year, are considered *succinct* constraints. A constraint is **succinct** if we can enumerate all and only those sequences that are guaranteed to satisfy the constraint, even before support counting begins. Suppose, here, $T = 2005$. By selecting the data for which $year = 2005$, we can enumerate all of the sequences *guaranteed to satisfy* the constraint before mining begins. In other words, we don't need to generate and test. Thus, such constraints contribute toward efficiency in that they avoid the substantial overhead of the generate-and-test paradigm.

Durations may also be defined as being related to sets of partitioned sequences, such as every year, or every month after stock dips, or every two weeks before and after an earthquake. In such cases, *periodic patterns* (Section 8.3.4) can be discovered.

Second, the constraint may be related to an **event folding window**, $w$. A set of events occurring within a specified period can be viewed as occurring together. If $w$ is set to be as long as the duration, $T$, it finds time-insensitive frequent patterns—these are essentially frequent patterns, such as "*In 1999, customers who bought a PC bought a digital camera as well*" (i.e., without bothering about which items were bought first). If $w$ is set to 0 (i.e., no event sequence folding), sequential patterns are found where each event occurs at a distinct time instant, such as "*A customer who bought a PC and then a digital camera is likely to buy an SD memory chip in a month.*" If $w$ is set to be something in between (e.g., for transactions occurring within the same month or within a sliding window of 24 hours), then these transactions are considered as occurring within the same period, and such sequences are "folded" in the analysis.

Third, a desired (time) **gap** between events in the discovered patterns may be specified as a constraint. Possible cases are: (1) $gap = 0$ (no gap is allowed), which is to find strictly consecutive sequential patterns like $a_{i-1}a_ia_{i+1}$. For example, if the event folding window is set to a week, this will find frequent patterns occurring in consecutive weeks; (2) $min\_gap \leq gap \leq max\_gap$, which is to find patterns that are separated by at least $min\_gap$ but at most $max\_gap$, such as "*If a person rents movie A, it is likely she will rent movie B within 30 days*" implies $gap \leq 30$ (days); and (3) $gap = c \neq 0$, which is to find patterns with an exact gap, $c$. It is straightforward to push gap constraints into the sequential pattern mining process. With minor modifications to the mining process, it can handle constraints with approximate gaps as well.

Finally, a user can specify constraints on the kinds of sequential patterns by providing "pattern templates" in the form of *serial episodes* and *parallel episodes* using *regular expressions*. A **serial episode** is a set of events that occurs in a total order, whereas a **parallel episode** is a set of events whose occurrence ordering is trivial. Consider the following example.

**Example 8.13** **Specifying serial episodes and parallel episodes with regular expressions.** Let the notation $(E,t)$ represent *event type E* at *time t*. Consider the data $(A,1)$, $(C,2)$, and $(B,5)$ with an event folding window width of $w = 2$, where the serial episode $A \rightarrow B$ and the parallel episode $A$ & $C$ both occur in the data. The user can specify constraints in the form of a regular expression, such as $(A|B)C*(D|E)$, which indicates that the user would like to find patterns where event $A$ and $B$ first occur (but they are parallel in that their relative ordering is unimportant), followed by one or a set of events $C$, followed by the events $D$ and $E$ (where $D$ can occur either before or after $E$). Other events can occur in between those specified in the regular expression. ■

A regular expression constraint may be neither antimonotonic nor monotonic. In such cases, we cannot use it to prune the search space in the same ways as described above. However, by modifying the PrefixSpan-based pattern-growth approach, such constraints can be handled elegantly. Let's examine one such example.

**Example 8.14** **Constraint-based sequential pattern mining with a regular expression constraint.** Suppose that our task is to mine sequential patterns, again using the sequence database, $S$, of Table 8.1. This time, however, we are particularly interested in patterns that match the regular expression constraint, $C = \langle a \star \{bb|(bc)d|dd\}\rangle$, with minimum support.

Such a regular expression constraint is neither antimonotonic, nor monotonic, nor succinct. Therefore, it cannot be pushed deep into the mining process. Nonetheless, this constraint can easily be integrated with the pattern-growth mining process as follows.

First, only the $\langle a\rangle$-projected database, $S|_{\langle a\rangle}$, needs to be mined, since the regular expression constraint $C$ starts with $a$. Retain only the sequences in $S|_{\langle a\rangle}$ that contain items within the set $\{b, c, d\}$. Second, the remaining mining can proceed from the suffix. This is essentially the *SuffixSpan* algorithm, which is symmetric to PrefixSpan in that it grows suffixes from the end of the sequence forward. The growth should match the suffix as the constraint, $\langle\{bb|(bc)d|dd\}\rangle$. For the projected databases that match these suffixes, we can grow sequential patterns either in prefix- or suffix-expansion manner to find all of the remaining sequential patterns. ■

Thus, we have seen several ways in which constraints can be used to improve the efficiency and usability of sequential pattern mining.

### 8.3.4 Periodicity Analysis for Time-Related Sequence Data

*"What is periodicity analysis?"* **Periodicity analysis** is the mining of periodic patterns, that is, the search for recurring patterns in time-related sequence data. Periodicity analysis can be applied to many important areas. For example, seasons, tides, planet trajectories, daily power consumptions, daily traffic patterns, and weekly TV programs all present certain periodic patterns. Periodicity analysis is often performed over time-series data, which consists of sequences of values or events typically measured at equal time intervals (e.g., hourly, daily, weekly). It can also be applied to other time-related sequence data where the value or event may occur at a nonequal time interval or at any time (e.g., on-line transactions). Moreover, the items to be analyzed can be numerical data, such as daily temperature or power consumption fluctuations, or categorical data (events), such as purchasing a product or watching a game.

The problem of mining periodic patterns can be viewed from different perspectives. Based on the coverage of the pattern, we can categorize periodic patterns into *full* versus *partial* periodic patterns:

- A **full periodic pattern** is a pattern where every point in time contributes (precisely or approximately) to the cyclic behavior of a time-related sequence. For example, all of the days in the year *approximately* contribute to the season cycle of the year.
- A **partial periodic pattern** specifies the periodic behavior of a time-related sequence at some but not all of the points in time. For example, Sandy reads the *New York Times* from 7:00 to 7:30 every weekday morning, but her activities at other times do not have much regularity. Partial periodicity is a looser form of periodicity than full periodicity and occurs more commonly in the real world.

Based on the precision of the periodicity, a pattern can be either *synchronous* or *asynchronous*, where the former requires that an event occur at a relatively fixed offset in each "stable" period, such as 3 p.m. every day, whereas the latter allows that the event fluctuates in a somewhat loosely defined period. A pattern can also be either *precise* or *approximate*, depending on the data value or the offset within a period. For example, if

Sandy reads the newspaper at 7:00 on some days, but at 7:10 or 7:15 on others, this is an approximate periodic pattern.

Techniques for full periodicity analysis for numerical values have been studied in signal analysis and statistics. Methods like FFT (Fast Fourier Transformation) are commonly used to transform data from the time domain to the frequency domain in order to facilitate such analysis.

Mining partial, categorical, and asynchronous periodic patterns poses more challenging problems in regards to the development of efficient data mining solutions. This is because most statistical methods or those relying on time-to-frequency domain transformations are either inapplicable or expensive at handling such problems.

Take mining partial periodicity as an example. Because partial periodicity mixes periodic events and nonperiodic events together in the same period, a time-to-frequency transformation method, such as FFT, becomes ineffective because it treats the time series as an inseparable flow of values. Certain periodicity detection methods can uncover some partial periodic patterns, but only if the period, length, and timing of the segment (subsequence of interest) in the partial patterns have certain behaviors and are explicitly specified. For the newspaper reading example, we need to explicitly specify details such as "Find the regular activities of Sandy during the half-hour after 7:00 for a period of 24 hours." A naïve adaptation of such methods to the partial periodic pattern mining problem would be prohibitively expensive, requiring their application to a huge number of possible combinations of the three parameters of period, length, and timing.

Most of the studies on mining partial periodic patterns apply the Apriori property heuristic and adopt some variations of Apriori-like mining methods. Constraints can also be pushed deep into the mining process. Studies have also been performed on the efficient mining of partially periodic event patterns or asynchronous periodic patterns with unknown or with approximate periods.

Mining partial periodicity may lead to the discovery of **cyclic or periodic association rules,** which are rules that associate a set of events that occur periodically. An example of a periodic association rule is "*Based on day-to-day transactions, if afternoon tea is well received between 3:00 to 5:00 p.m., dinner will sell well between 7:00 to 9:00 p.m. on weekends.*"

Due to the diversity of applications of time-related sequence data, further development of efficient algorithms for mining various kinds of periodic patterns in sequence databases is desired.

## 8.4 Mining Sequence Patterns in Biological Data

Bioinformatics is a promising young field that applies computer technology in molecular biology and develops algorithms and methods to manage and analyze biological data. Because DNA and protein sequences are essential biological data and exist in huge volumes as well, it is important to develop effective methods to compare and align biological sequences and discover biosequence patterns.

Before we get into further details, let's look at the type of data being analyzed. DNA and proteins sequences are long linear chains of chemical components. In the case of DNA, these components or "building blocks" are four **nucleotides** (also called *bases*), namely adenine (A), cytosine (C), guanine (G), and thymine (T). In the case of proteins, the components are 20 **amino acids**, denoted by 20 different letters of the alphabet. A gene is a sequence of typically hundreds of individual nucleotides arranged in a particular order. A **genome** is the complete set of genes of an organism. When proteins are needed, the corresponding genes are transcribed into RNA. RNA is a chain of nucleotides. DNA directs the synthesis of a variety of RNA molecules, each with a unique role in cellular function.

"*Why is it useful to compare and align biosequences?*" The alignment is based on the fact that all living organisms are related by evolution. This implies that the nucleotide (DNA, RNA) and proteins sequences of the species that are closer to each other in evolution should exhibit more similarities. An **alignment** is the process of lining up sequences to achieve a maximal level of identity, which also expresses the degree of similarity between sequences. Two sequences are **homologous** if they share a common ancestor. The degree of similarity obtained by sequence alignment can be useful in determining the possibility of homology between two sequences. Such an alignment also helps determine the relative positions of multiple species in an evolution tree, which is called a **phylogenetic tree**.

In Section 8.4.1, we first study methods for *pairwise alignment* (i.e., the alignment of two biological sequences). This is followed by methods for *multiple sequence alignment*. Section 8.4.2 introduces the popularly used Hidden Markov Model (HMM) for biological sequence analysis.

### 8.4.1 Alignment of Biological Sequences

The problem of alignment of biological sequences can be described as follows: *Given two or more input biological sequences, identify similar sequences with long conserved subsequences*. If the number of sequences to be aligned is exactly two, it is called **pairwise sequence alignment**; otherwise, it is **multiple sequence alignment**. The sequences to be compared and aligned can be either nucleotides (DNA/RNA) or amino acids (proteins). For nucleotides, two symbols align if they are identical. However, for amino acids, two symbols align if they are identical, or if one can be derived from the other by substitutions that are likely to occur in nature. There are two kinds of alignments: *local alignments* versus *global alignments*. The former means that only portions of the sequences are aligned, whereas the latter requires alignment over the entire length of the sequences.

For either nucleotides or amino acids, insertions, deletions, and substitutions occur in nature with different probabilities. **Substitution matrices** are used to represent the probabilities of substitutions of nucleotides or amino acids and probabilities of insertions and deletions. Usually, we use the gap character, "−", to indicate positions where it is preferable not to align two symbols. To evaluate the quality of alignments, a *scoring* mechanism is typically defined, which usually counts identical or similar symbols as positive scores and gaps as negative ones. The algebraic sum of the scores is taken as the alignment measure. The goal of alignment is to achieve the maximal score among all the possible alignments. However, it is very expensive (more exactly, an NP-hard problem) to find optimal alignment. Therefore, various heuristic methods have been developed to find suboptimal alignments.

## Pairwise Alignment

**Example 8.15** **Pairwise alignment.** Suppose we have two amino acid sequences as follows, and the substitution matrix of amino acids for pairwise alignment is shown in Table 8.3.

Suppose the penalty for initiating a gap (called the *gap penalty*) is −8 and that for extending a gap (i.e., *gap extension penalty*) is also −8. We can then compare two potential sequence alignment candidates, as shown in Figure 8.8 (a) and (b) by calculating their total alignment scores.

The total score of the alignment for Figure 8.8(a) is $(-2) + (-8) + (5) + (-8) + (-8) + (15) + (-8) + (10) + (6) + (-8) + (6) = 0$, whereas that for Figure 8.8(b) is

**Table 8.3** The substitution matrix of amino acids.

*HEAGAWGHEE*

*PAWHEAE*

| | *A* | *E* | *G* | *H* | *W* |
|---|---|---|---|---|---|
| *A* | 5 | −1 | 0 | −2 | −3 |
| *E* | −1 | 6 | −3 | 0 | −3 |
| *H* | −2 | 0 | −2 | 10 | −3 |
| *P* | −1 | −1 | −2 | −2 | −4 |
| *W* | −3 | −3 | −3 | −3 | 15 |

```
H  E  A  G  A  W  G  H  E  –  E
      |        |     |  |     |
P  –  A  –  –  W  –  H  E  A  E
              (a)
H  E  A  G  A  W  G  H  E  –  E
            |  |     |  |     |
–  –  P  –  A  W  –  H  E  A  E
              (b)
```

**Figure 8.8** Scoring two potential pairwise alignments, (*a*) and (*b*), of amino acids.

$(-8)+(-8)+(-1)+(-8)+(5)+(15)+(-8)+(10)+(6)+(-8)+(6)=1$. Thus the alignment of Figure 8.8(b) is slightly better than that in Figure 8.8(a). ■

Biologists have developed $20 \times 20$ triangular matrices that provide the weights for comparing identical and different amino acids as well as the penalties that should be attributed to gaps. Two frequently used matrices are PAM (Percent Accepted Mutation) and BLOSUM (BlOcks SUbstitution Matrix). These substitution matrices represent the weights obtained by comparing the amino acid substitutions that have occurred through evolution.

For global pairwise sequence alignment, two influential algorithms have been proposed: the *Needleman-Wunsch Algorithm* and the *Smith-Waterman Algorithm*. The former uses weights for the outmost edges that encourage the best overall global alignment, whereas the latter favors the contiguity of segments being aligned. Both build up "optimal" alignment from "optimal" alignments of subsequences. Both use the methodology of dynamic programming. Since these algorithms use recursion to fill in an intermediate results table, it takes $O(mn)$ space and $O(n^2)$ time to execute them. Such computational complexity could be feasible for moderate-sized sequences but is not feasible for aligning large sequences, especially for entire genomes, where a *genome* is the complete set of genes of an organism. Another approach called *dot matrix plot* uses Boolean matrices to represent possible alignments that can be detected visually. The method is simple and facilitates easy visual inspection. However, it still takes $O(n^2)$ in time and space to construct and inspect such matrices.

To reduce the computational complexity, heuristic alignment algorithms have been proposed. Heuristic algorithms speed up the alignment process at the price of possibly missing the best scoring alignment. There are two influential heuristic alignment programs: (1) BLAST (Basic Local Alignment Search Tool), and (2) FASTA (Fast Alignment Tool). Both find high-scoring local alignments between a query sequence and a target database. Their basic idea is to first locate high-scoring short stretches and then extend them to achieve suboptimal alignments. Because the BLAST algorithm has been very popular in biology and bioinformatics research, we examine it in greater detail here.

## The BLAST Local Alignment Algorithm

The **BLAST** algorithm was first developed by Altschul, Gish, Miller, et al. around 1990 at the National Center for Biotechnology Information (NCBI). The software, its tutorials, and a wealth of other information can be accessed at *www.ncbi.nlm.nih.gov/BLAST/*. BLAST finds regions of local similarity between biosequences. The program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches. BLAST can be used to infer functional and evolutionary relationships between sequences as well as to help identify members of gene families.

The NCBI website contains many common BLAST databases. According to their content, they are grouped into nucleotide and protein databases. NCBI also provides specialized BLAST databases such as the vector screening database, a variety of genome databases for different organisms, and trace databases.

BLAST applies a heuristic method to find the highest local alignments between a query sequence and a database. BLAST improves the overall speed of search by breaking the sequences to be compared into sequences of fragments (referred to as **words**) and initially seeking matches between these words. In BLAST, the words are considered as $k$-tuples. For DNA nucleotides, a word typically consists of 11 bases (nucleotides), whereas for proteins, a word typically consists of 3 amino acids. BLAST first creates a hash table of neighborhood (i.e., closely matching) words, while the threshold for "closeness" is set based on statistics. It starts from exact matches to neighborhood words. Because good alignments should contain many close matches, we can use statistics to determine which matches are significant. By hashing, we can find matches in $O(n)$ (linear) time. By extending matches in both directions, the method finds high-quality alignments consisting of many high-scoring and maximum segment pairs.

There are many versions and extensions of the BLAST algorithms. For example, MEGABLAST, Discontiguous MEGABLAST, and BLASTN all can be used to identify a nucleotide sequence. MEGABLAST is specifically designed to efficiently find long alignments between very similar sequences, and thus is the best tool to use to find the identical match to a query sequence. Discontiguous MEGABLAST is better at finding nucleotide sequences that are similar, but not identical (i.e., gapped alignments), to a nucleotide query. One of the important parameters governing the sensitivity of BLAST searches is the length of the initial words, or *word size*. The word size is adjustable in BLASTN and can be reduced from the default value to a minimum of 7 to increase search sensitivity. Thus BLASTN is better than MEGABLAST at finding alignments to related nucleotide sequences from other organisms. For protein searches, BLASTP, PSI-BLAST, and PHI-BLAST are popular. Standard protein-protein BLAST (BLASTP) is used for both identifying a query amino acid sequence and for finding similar sequences in protein databases. Position-Specific Iterated (PSI)-BLAST is designed for more sensitive protein-protein similarity searches. It is useful for finding very distantly related proteins. Pattern-Hit Initiated (PHI)-BLAST can do a restricted protein pattern search. It is designed to search for proteins that contain a pattern specified by the user and are similar to the query sequence in the vicinity of the pattern. This dual requirement is intended to reduce the number of database hits that contain the pattern, but are likely to have no true homology to the query.

## Multiple Sequence Alignment Methods

Multiple sequence alignment is usually performed on a set of sequences of amino acids that are believed to have similar structures. The goal is to find common patterns that are conserved among all the sequences being considered.

The alignment of multiple sequences has many applications. First, such an alignment may assist in the identification of highly conserved residues (amino acids), which are likely to be essential sites for structure and function. This will guide or help pairwise alignment as well. Second, it will help build gene or protein families using conserved regions, forming a basis for phylogenetic analysis (i.e., the inference of evolutionary relationships between genes). Third, conserved regions can be used to develop primers for amplifying DNA sequences and probes for DNA microarray analysis.

From the computational point of view, it is more challenging to align multiple sequences than to perform pairwise alignment of two sequences. This is because multisequence alignment can be considered as a multidimensional alignment problem, and there are many more possibilities for approximate alignments of subsequences in multiple dimensions.

There are two major approaches for approximate multiple sequence alignment. The first method reduces a multiple alignment to a series of pairwise alignments and then combines the result. The popular **Feng-Doolittle alignment** method belongs to this approach. Feng-Doolittle alignment first computes all of the possible pairwise alignments by dynamic programming and converts or normalizes alignment scores to distances. It then constructs a "guide tree" by clustering and performs progressive alignment based on the guide tree in a bottom-up manner. Following this approach, a multiple alignment tool, Clustal W, and its variants have been developed as software packages for multiple sequence alignments. The software handles a variety of input/output formats and provides displays for visual inspection.

The second multiple sequence alignment method uses hidden Markov models (HMMs). Due to the extensive use and popularity of hidden Markov models, we devote an entire section to this approach. It is introduced in Section 8.4.2, which follows.

From the above discussion, we can see that several interesting methods have been developed for multiple sequence alignment. Due to its computational complexity, the development of effective and scalable methods for multiple sequence alignment remains an active research topic in biological data mining.

### 8.4.2 Hidden Markov Model for Biological Sequence Analysis

Given a biological sequence, such as a DNA sequence or an amino acid (protein), biologists would like to analyze what that sequence represents. For example, is a given DNA sequence a gene or not? Or, to which family of proteins does a particular amino acid sequence belong? In general, given sequences of symbols from some alphabet, we would like to represent the structure or statistical regularities of classes of sequences. In this section, we discuss *Markov chains* and *hidden Markov models*—probabilistic models that are well suited for this type of task. Other areas of research, such as speech and pattern recognition, are faced with similar sequence analysis tasks.

To illustrate our discussion of Markov chains and hidden Markov models, we use a classic problem in biological sequence analysis—that of finding *CpG islands* in a DNA sequence. Here, the alphabet consists of four **nucleotides**, namely, A (adenine), C (cytosine), G (guanine), and T (thymine). **CpG** denotes a pair (or subsequence) of nucleotides, where G appears immediately after C along a DNA strand. The C in a CpG pair is often modified by a process known as *methylation* (where the C is replaced by methyl-C, which tends to mutate to T). As a result, CpG pairs occur infrequently in the human genome. However, methylation is often suppressed around *promotors* or "start" regions of many genes. These areas contain a relatively high concentration of CpG pairs, collectively referred to along a chromosome as **CpG islands**, which typically vary in length from a few hundred to a few thousand nucleotides long. CpG islands are very useful in genome mapping projects.

Two important questions that biologists have when studying DNA sequences are (1) given a short sequence, is it from a CpG island or not? and (2) given a long sequence, can we find all of the CpG islands within it? We start our exploration of these questions by introducing Markov chains.

## Markov Chain

A **Markov chain** is a model that generates sequences in which the probability of a symbol depends only on the previous symbol. Figure 8.9 is an example Markov chain model. A Markov chain model is defined by (a) a set of *states*, $Q$, which emit symbols and (b) a set of *transitions* between states. States are represented by circles and transitions are represented by arrows. Each transition has an associated **transition probability**, $a_{ij}$, which represents the conditional probability of going to state $j$ in the next step, given that the current state is $i$. The sum of all transition probabilities from a given state must equal 1, that is, $\sum_{j \in Q} a_{ij} = 1$ for all $j \in Q$. If an arc is not shown, it is assumed to have a 0 probability. The transition probabilities can also be written as a *transition matrix*, $A = \{a_{ij}\}$.

**Example 8.16** **Markov chain.** The Markov chain in Figure 8.9 is a probabilistic model for CpG islands. The states are A, C, G, and T. For readability, only some of the transition probabilities are shown. For example, the transition probability from state G to state T is 0.14, that is, $P(x_i = \text{T}|x_{i-1} = \text{G}) = 0.14$. Here, the emitted symbols are understood. For example, the symbol C is emitted when transitioning from state C. In speech recognition, the symbols emitted could represent spoken words or phrases. ■

Given some sequence $x$ of length $L$, how probable is $x$ given the model? If $x$ is a DNA sequence, we could use our Markov chain model to determine how probable it is that $x$ is from a CpG island. To do so, we look at the probability of $x$ as a *path*, $x_1x_2 \dots x_L$, in the chain. This is the probability of starting in the first state, $x_1$, and making successive transitions to $x_2, x_3$, and so on, to $x_L$. In a Markov chain model, the probability of $x_L$

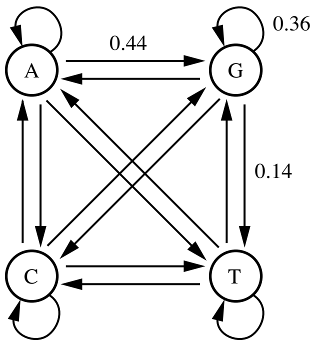

**Figure 8.9** A Markov chain model.

depends on the value of only the *previous one state*, $x_{L-1}$, not on the entire previous sequence.[9] This characteristic is known as the **Markov property**, which can be written as

$$\begin{aligned} P(x) &= P(x_L|x_{L-1})P(x_{L-1}|x_{L-2})\cdots P(x_2|x_1)P(x_1) \\ &= P(x_1)\prod_{i=2}^{L} P(x_i|x_{i-1}). \end{aligned} \tag{8.7}$$

That is, the Markov chain can only "remember" the previous one state of its history. Beyond that, it is "memoryless."

In Equation (8.7), we need to specify $P(x_1)$, the probability of the starting state. For simplicity, we would like to model this as a transition too. This can be done by adding a *begin* state, denoted 0, so that the starting state becomes $x_0 = 0$. Similarly, we can add an *end* state, also denoted as 0. Note that $P(x_i|x_{i-1})$ is the transition probability, $a_{x_{i-1}x_i}$. Therefore, Equation (8.7) can be rewritten as

$$P(x) = \prod_{i=1}^{L} a_{x_{i-1}x_i}, \tag{8.8}$$

which computes the probability that sequence $x$ belongs to the given Markov chain model, that is, $P(x|model)$. Note that the begin and end states are silent in that they do not emit symbols in the path through the chain.

We can use the Markov chain model for classification. Suppose that we want to distinguish CpG islands from other "non-CpG" sequence regions. Given training sequences from CpG islands (labeled "+") and from non-CpG islands (labeled "−"), we can construct two Markov chain models—the first, denoted "+", to represent CpG islands, and the second, denoted "−", to represent non-CpG islands. Given a sequence, $x$, we use the respective models to compute $P(x|+)$, the probability that $x$ is from a CpG island, and $P(x|-)$, the probability that it is from a non-CpG island. The *log-odds ratio* can then be used to classify $x$ based on these two probabilities.

*"But first, how can we estimate the transition probabilities for each model?"* Before we can compute the probability of $x$ being from either of the two models, we need to estimate the transition probabilities for the models. Given the CpG (+) training sequences, we can estimate the transition probabilities for the CpG island model as

$$a_{ij}^{+} = \frac{c_{ij}^{+}}{\sum_k c_{ik}^{+}}, \tag{8.9}$$

where $c_{ij}^{+}$ is the number of times that nucleotide $j$ follows nucleotide $i$ in the given sequences labeled "+". For the non-CpG model, we use the non-CpG island sequences (labeled "−") in a similar way to estimate $a_{ij}^{-}$.

[9]This is known as a **first-order Markov chain model**, since $x_L$ depends only on the previous state, $x_{L-1}$. In general, for the $k$-**th-order Markov chain model**, the probability of $x_L$ depends on the values of only the *previous k* states.

To determine whether $x$ is from a CpG island or not, we compare the models using the **logs-odds ratio**, defined as

$$log\frac{P(x|+)}{P(x|-)} = \sum_{i=1}^{L} log\frac{a^{+}_{x_{i-1}x_i}}{a^{-}_{x_{i-1}x_i}}. \tag{8.10}$$

If this ratio is greater than 0, then we say that $x$ is from a CpG island.

**Example 8.17** **Classification using a Markov chain.** Our model for CpG islands and our model for non-CpG islands both have the same structure, as shown in our example Markov chain of Figure 8.9. Let $CpG^{+}$ be the transition matrix for the CpG island model. Similarly, $CpG^{-}$ is the transition matrix for the non-CpG island model. These are (adapted from Durbin, Eddy, Krogh, and Mitchison [DEKM98]):

$$CpG^{+} = \left[\begin{array}{ccccc} & A & C & G & T \\ A & 0.20 & 0.26 & 0.44 & 0.10 \\ C & 0.16 & 0.36 & 0.28 & 0.20 \\ G & 0.15 & 0.35 & 0.36 & 0.14 \\ T & 0.09 & 0.37 & 0.36 & 0.18 \end{array}\right] \tag{8.11}$$

$$CpG^{-} = \left[\begin{array}{ccccc} & A & C & G & T \\ A & 0.27 & 0.19 & 0.31 & 0.23 \\ C & 0.33 & 0.31 & 0.08 & 0.28 \\ G & 0.26 & 0.24 & 0.31 & 0.19 \\ T & 0.19 & 0.25 & 0.28 & 0.28 \end{array}\right] \tag{8.12}$$

Notice that the transition probability $a^{+}_{CG} = 0.28$ is higher than $a^{-}_{CG} = 0.08$. Suppose we are given the sequence $x$ = CGCG. The log-odds ratio of $x$ is

$$log\frac{0.28}{0.08} + log\frac{0.35}{0.24} + log\frac{0.28}{0.08} = 1.25 > 0.$$

Thus, we say that $x$ is from a CpG island. ■

In summary, we can use a Markov chain model to determine if a DNA sequence, $x$, is from a CpG island. This was the first of our two important questions mentioned at the beginning of this section. To answer the second question, that of finding all of the CpG islands in a given sequence, we move on to hidden Markov models.

## Hidden Markov Model

Given a long DNA sequence, how can we find all CpG islands within it? We could try the Markov chain method above, using a sliding window. For each window, we could compute the log-odds ratio. CpG islands within intersecting windows could be merged to determine CpG islands within the long sequence. This approach has some difficulties: It is not clear what window size to use, and CpG islands tend to vary in length.

What if, instead, we merge the two Markov chains from above (for CpG islands and non-CpG islands, respectively) and add transition probabilities between the two chains? The result is a *hidden Markov model*, as shown in Figure 8.10. The states are renamed by adding "+" and "−" labels to distinguish them. For readability, only the transitions between "+" and "−" states are shown, in addition to those for the begin and end states. Let $\pi = \pi_1\pi_2 \ldots \pi_L$ be a path of states that generates a sequence of symbols, $x = x_1x_2 \ldots x_L$. In a Markov chain, the path through the chain for $x$ is unique. With a hidden Markov model, however, different paths can generate the same sequence. For example, the states $C^+$ and $C^-$ both emit the symbol C. Therefore, we say the model is "hidden" in that we do not know for sure which states were visited in generating the sequence. The transition probabilities between the original two models can be determined using training sequences containing transitions between CpG islands and non-CpG islands.

**A Hidden Markov Model** (HMM) is defined by

- a set of states, $Q$
- a set of transitions, where transition probability $a_{kl} = P(\pi_i = l|\pi_{i-1} = k)$ is the probability of transitioning from state $k$ to state $l$ for $k, l \in Q$
- an **emission probability**, $e_k(b) = P(x_i = b|\pi_i = k)$, for each state, $k$, and each symbol, $b$, where $e_k(b)$ is the probability of seeing symbol $b$ in state $k$. The sum of all emission probabilities at a given state must equal 1, that is, $\sum_b e_k = 1$ for each state, $k$.

**Example 8.18** **A hidden Markov model.** The transition matrix for the hidden Markov model of Figure 8.10 is larger than that of Example 8.16 for our earlier Markov chain example.

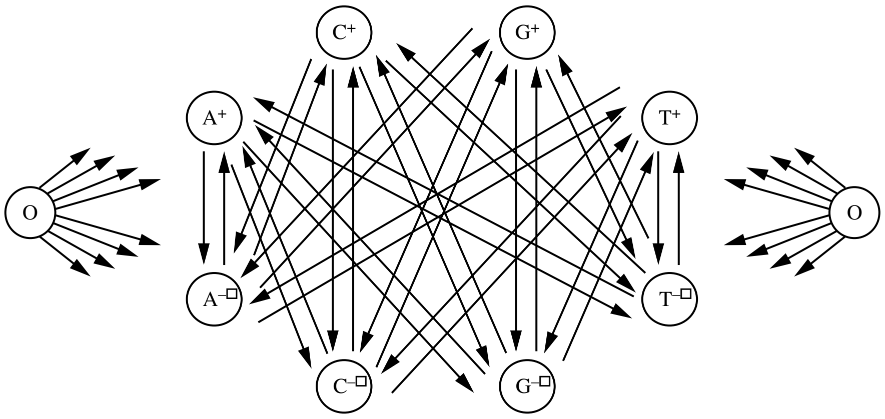

**Figure 8.10** A hidden Markov model.

It contains the states $A^+$, $C^+$, $G^+$, $T^+$, $A^-$, $C^-$, $G^-$, $T^-$ (not shown). The transition probabilities between the "+" states are as before. Similarly, the transition probabilities between the "−" states are as before. The transition probabilities between "+" and "−" states can be determined as mentioned above, using training sequences containing known transitions from CpG islands to non-CpG islands, and vice versa. The emission probabilities are $e_{A^+}(A) = 1$, $e_{A^+}(C) = 0$, $e_{A^+}(G) = 0$, $e_{A^+}(T) = 0$, $e_{A^-}(A) = 1$, $e_{A^-}(C) = 0$, $e_{A^-}(G) = 0$, $e_{A^-}(T) = 0$, and so on. Although here the probability of emitting a symbol at a state is either 0 or 1, in general, emission probabilities need not be zero-one. ■

Tasks using hidden Markov models include:

- *Evaluation*: Given a sequence, $x$, determine the probability, $P(x)$, of obtaining $x$ in the model.
- *Decoding:* Given a sequence, determine the most probable path through the model that produced the sequence.
- *Learning*: Given a model and a set of training sequences, find the model parameters (i.e., the transition and emission probabilities) that explain the training sequences with relatively high probability. The goal is to find a model that generalizes well to sequences we have not seen before.

Evaluation, decoding, and learning can be handled using the forward algorithm, Viterbi algorithm, and Baum-Welch algorithm, respectively. These algorithms are discussed in the following sections.

## Forward Algorithm

What is the probability, $P(x)$, that sequence $x$ was generated by a given hidden Markov model (where, say, the model represents a sequence class)? This problem can be solved using the **forward algorithm**.

Let $x = x_1x_2 \dots x_L$ be our sequence of symbols. A path is a sequence of states. Many paths can generate $x$. Consider one such path, which we denote $\pi = \pi_1\pi_2 \dots \pi_L$. If we incorporate the begin and end states, denoted as 0, we can write $\pi$ as $\pi_0 = 0, \pi_1\pi_2 \dots \pi_L, \pi_{L+1} = 0$. The probability that the model generated sequence $x$ using path $\pi$ is

$$\begin{aligned} P(x, \pi) &= a_{0\pi_1} e_{\pi_1}(x_1) \cdot a_{\pi_1\pi_2} e_{\pi_2}(x_2) \cdot \dots \cdot a_{\pi_{L-1}\pi_L} e_{\pi_L}(x_L) \cdot a_{\pi_L 0} \\ &= a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) a_{\pi_i\pi_{i+1}} \end{aligned} \tag{8.13}$$

where $\pi_{L+1} = 0$. We must, however, consider all of the paths that can generate $x$. Therefore, the probability of $x$ given the model is

$$P(x) = \sum_{\pi} P(x, \pi). \tag{8.14}$$

That is, we add the probabilities of all possible paths for $x$.

```
Algorithm: Forward algorithm. Find the probability, $P(x)$, that sequence $x$ was generated by the given hidden
  Markov model.

Input:
  ■ A hidden Markov model, defined by a set of states, $Q$, that emit symbols, and by transition and emission
    probabilities;
  ■ $x$, a sequence of symbols.

Output: Probability, $P(x)$.

Method:
  (1) Initialization ($i = 0$):    $f_0(0) = 1$, $f_k(0) = 0$ for $k > 0$
  (2) Recursion ($i = 1 \ldots L$): $f_l(i) = e_l(x_i) \sum_k f_k(i-1) a_{kl}$
  (3) Termination:                 $P(x) = \sum_k f_k(L) a_{k0}$
```

**Figure 8.11** Forward algorithm.

Unfortunately, the number of paths can be exponential with respect to the length, $L$, of $x$, so brute force evaluation by enumerating all paths is impractical. The forward algorithm exploits a dynamic programming technique to solve this problem. It defines **forward variables**, $f_k(i)$, to be the probability of being in state $k$ having observed the first $i$ symbols of sequence $x$. We want to compute $f_{\pi_{L+1=0}}(L)$, the probability of being in the end state having observed all of sequence $x$.

The forward algorithm is shown in Figure 8.11. It consists of three steps. In step 1, the forward variables are initialized for all states. Because we have not viewed any part of the sequence at this point, the probability of being in the start state is 1 (i.e., $f_0(0) = 1$), and the probability of being in any other state is 0. In step 2, the algorithm sums over all the probabilities of all the paths leading from one state emission to another. It does this recursively for each move from state to state. Step 3 gives the termination condition. The whole sequence (of length $L$) has been viewed, and we enter the end state, 0. We end up with the summed-over probability of generating the required sequence of symbols.

## Viterbi Algorithm

Given a sequence, $x$, what is the most probable path in the model that generates $x$? This problem of decoding can be solved using the **Viterbi algorithm**.

Many paths can generate $x$. We want to find the most probable one, $\pi^\star$, that is, the path that maximizes the probability of having generated $x$. This is $\pi^\star = argmax_\pi P(\pi|x)$.[10] It so happens that this is equal to $argmax_\pi P(x, \pi)$. (The proof is left as an exercise for the reader.) We saw how to compute $P(x, \pi)$ in Equation (8.13). For a sequence of length $L$, there are $|Q|^L$ possible paths, where $|Q|$ is the number of states in the model. It is

[10]In mathematics, *argmax* stands for the argument of the maximum. Here, this means that we want the path, $\pi$, for which $P(\pi|x)$ attains its maximum value.

infeasible to enumerate all of these possible paths! Once again, we resort to a dynamic programming technique to solve the problem.

At each step along the way, the Viterbi algorithm tries to find the most probable path leading from one symbol of the sequence to the next. We define $v_l(i)$ to be the probability of the most probable path accounting for the first $i$ symbols of $x$ and ending in state $l$. To find $\pi^\star$, we need to compute $max_k v_k(L)$, the probability of the most probable path accounting for all of the sequence and ending in the end state. The probability, $v_l(i)$, is

$$v_l(i) = e_l(x_i) \cdot max_k(v_l(k)a_{kl}), \tag{8.15}$$

which states that the most probable path that generates $x_1 \ldots x_i$ and ends in state $l$ has to emit $x_i$ in state $x_l$ (hence, the emission probability, $e_l(x_i)$) and has to contain the most probable path that generates $x_1 \ldots x_{i-1}$ and ends in state $k$, followed by a transition from state $k$ to state $l$ (hence, the transition probability, $a_{kl}$). Thus, we can compute $v_k(L)$ for any state, $k$, recursively to obtain the probability of the most probable path.

The Viterbi algorithm is shown in Figure 8.12. Step 1 performs initialization. Every path starts at the begin state (0) with probability 1. Thus, for $i = 0$, we have $v_0(0) = 1$, and the probability of starting at any other state is 0. Step 2 applies the *recurrence formula* for $i = 1$ to $L$. At each iteration, we assume that we know the most likely path for $x_1 \ldots x_{i-1}$ that ends in state $k$, for all $k \in Q$. To find the most likely path to the $i$-th state from there, we maximize $v_k(i-1)a_{kl}$ over all predecessors $k \in Q$ of $l$. To obtain $v_l(i)$, we multiply by $e_l(x_i)$ since we have to emit $x_i$ from $l$. This gives us the first formula in step 2. The values $v_k(i)$ are stored in a $Q \times L$ dynamic programming matrix. We keep pointers (*ptr*) in this matrix so that we can obtain the path itself. The algorithm terminates in step 3, where finally, we have $max_k v_k(L)$. We enter the end state of 0 (hence, the transition probability, $a_{k0}$) but do not emit a symbol. The Viterbi algorithm runs in $O(|Q|^2|L|)$ time. It is more efficient than the forward algorithm because it investigates only the most probable path and avoids summing over all possible paths.

**Algorithm: Viterbi algorithm.** Find the most probable path that emits the sequence of symbols, $x$.

**Input:**

- A hidden Markov model, defined by a set of states, $Q$, that emit symbols, and by transition and emission probabilities;
- $x$, a sequence of symbols.

**Output:** The most probable path, $\pi^*$.

**Method:**

```
(1) Initialization (i = 0):   $v_0(0) = 1, v_k(0) = 0$ for $k > 0$
(2) Recursion (i = 1...L):    $v_l(i) = e_l(x_i) max_k(v_k(i-1)a_{kl})$
                              $ptr_i(l) = argmax_k(v_k(i-1)a_{kl})$
(3) Termination:              $P(x, \pi^*) = max_k(v_k(L)a_{k0})$
                              $\pi^*_L = argmax_k(v_k(L)a_{k0})$
```

**Figure 8.12** Viterbi (decoding) algorithm.

## Baum-Welch Algorithm

Given a training set of sequences, how can we determine the parameters of a hidden Markov model that will best explain the sequences? In other words, we want to learn or adjust the transition and emission probabilities of the model so that it can predict the path of future sequences of symbols. If we know the state path for each training sequence, learning the model parameters is simple. We can compute the percentage of times each particular transition or emission is used in the set of training sequences to determine $a_{kl}$, the transition probabilities, and $e_k(b)$, the emission probabilities.

When the paths for the training sequences are unknown, there is no longer a direct closed-form equation for the estimated parameter values. An iterative procedure must be used, like the **Baum-Welch algorithm**. The Baum-Welch algorithm is a special case of the EM algorithm (Section 7.8.1), which is a family of algorithms for learning probabilistic models in problems that involve hidden states.

The Baum-Welch algorithm is shown in Figure 8.13. The problem of finding the optimal transition and emission probabilities is intractable. Instead, the Baum-Welch algorithm finds a locally optimal solution. In step 1, it initializes the probabilities to an arbitrary estimate. It then continuously re-estimates the probabilities (step 2) until convergence (i.e., when there is very little change in the probability values between iterations). The re-estimation first calculates the expected transmission and emission probabilities. The transition and emission probabilities are then updated to maximize the likelihood of the expected values.

In summary, Markov chains and hidden Markov models are probabilistic models in which the probability of a state depends only on that of the previous state. They are particularly useful for the analysis of biological sequence data, whose tasks include evaluation, decoding, and learning. We have studied the forward, Viterbi, and Baum-Welch algorithms. The algorithms require multiplying many probabilities, resulting in very

**Algorithm: Baum-Welch algorithm.** Find the model parameters (transition and emission probabilities) that best explain the training set of sequences.

**Input:**

- A training set of sequences.

**Output:**

- Transition probabilities, $a_{kl}$;
- Emission probabilities, $e_k(b)$;

**Method:**

```
(1) initialize the transmission and emission probabilities;
(2) iterate until convergence
        (2.1) calculate the expected number of times each transition or emission is used
        (2.2) adjust the parameters to maximize the likelihood of these expected values
```

**Figure 8.13** Baum-Welch (learning) algorithm.

small numbers that can cause underflow arithmetic errors. A way around this is to use the logarithms of the probabilities.

## 8.5 Summary

- **Stream data** flow in and out of a computer system *continuously* and with varying update rates. They are *temporally ordered, fast changing, massive* (e.g., gigabytes to terabytes in volume), and *potentially infinite*. Applications involving stream data include telecommunications, financial markets, and satellite data processing.
- **Synopses** provide *summaries* of stream data, which typically can be used to return *approximate* answers to queries. Random sampling, sliding windows, histograms, multiresolution methods (e.g., for data reduction), sketches (which operate in a single pass), and randomized algorithms are all forms of synopses.
- The **tilted time frame** model allows data to be stored at multiple granularities of time. The most recent time is registered at the finest granularity. The most distant time is at the coarsest granularity.
- A **stream data cube** can store compressed data by (1) using the tilted time frame model on the time dimension, (2) storing data at only some **critical layers**, which reflect the levels of data that are of most interest to the analyst, and (3) performing *partial materialization* based on "popular paths" through the critical layers.
- Traditional methods of **frequent itemset mining, classification**, and **clustering** tend to scan the data multiple times, making them infeasible for stream data. Stream-based versions of such mining instead try to find approximate answers within a user-specified error bound. Examples include the Lossy Counting algorithm for frequent itemset stream mining; the Hoeffding tree, VFDT, and CVFDT algorithms for stream data classification; and the STREAM and CluStream algorithms for stream data clustering.
- A **time-series database** consists of sequences of values or events changing with time, typically measured at equal time intervals. Applications include stock market analysis, economic and sales forecasting, cardiogram analysis, and the observation of weather phenomena.
- **Trend analysis** decomposes time-series data into the following: *trend* (long-term) *movements*, *cyclic movements, seasonal movements* (which are systematic or calendar related), and *irregular movements* (due to random or chance events).
- **Subsequence matching** is a form of *similarity search* that finds subsequences that are similar to a given query sequence. Such methods match subsequences that have the same shape, while accounting for gaps (missing values) and differences in baseline/offset and scale.
- A **sequence database** consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. Examples of sequence data include customer shopping sequences, Web clickstreams, and biological sequences.

- **Sequential pattern mining** is the mining of frequently occurring ordered events or subsequences as patterns. Given a sequence database, any sequence that satisfies minimum support is **frequent** and is called a **sequential pattern**. An example of a sequential pattern is "*Customers who buy a Canon digital camera are likely to buy an HP color printer within a month*." Algorithms for sequential pattern mining include GSP, SPADE, and PrefixSpan, as well as CloSpan (which mines closed sequential patterns).
- **Constraint-based mining** of sequential patterns incorporates user-specified constraints to reduce the search space and derive only patterns that are of interest to the user. Constraints may relate to the *duration* of a sequence, to an *event folding window* (where events occurring within such a window of time can be viewed as occurring together), and to *gaps* between events. *Pattern templates* may also be specified as a form of constraint using regular expressions.
- **Periodicity analysis** is the mining of periodic patterns, that is, the search for recurring patterns in time-related sequence databases. *Full periodic* and *partial periodic* patterns can be mined, as well as *periodic association rules*.
- **Biological sequence analysis** compares, aligns, indexes, and analyzes biological sequences, which can be either sequences of nucleotides or of amino acids. Biosequence analysis plays a crucial role in bioinformatics and modern biology. Such analysis can be partitioned into two essential tasks: **pairwise sequence alignment** and **multiple sequence alignment**. The dynamic programming approach is commonly used for sequence alignments. Among many available analysis packages, BLAST (Basic Local Alignment Search Tool) is one of the most popular tools in biosequence analysis.
- **Markov chains** and **hidden Markov models** are probabilistic models in which the probability of a state depends only on that of the previous state. They are particularly useful for the analysis of biological sequence data. Given a sequence of symbols, $x$, the forward algorithm finds the probability of obtaining $x$ in the model, whereas the Viterbi algorithm finds the most probable path (corresponding to $x$) through the model. The Baum-Welch algorithm learns or adjusts the model parameters (*transition* and *emission* probabilities) so as to best explain a set of training sequences.

## Exercises

8.1 A *stream data cube* should be relatively stable in size with respect to infinite data streams. Moreover, it should be incrementally updateable with respect to infinite data streams. Show that the stream cube proposed in Section 8.1.2 satisfies these two requirements.

8.2 In stream data analysis, we are often interested in only the nontrivial or exceptionally large cube cells. These can be formulated as *iceberg conditions*. Thus, it may seem that the iceberg cube [BR99] is a likely model for stream cube architecture. Unfortunately, this is not the case because iceberg cubes cannot accommodate the incremental updates required due to the constant arrival of new data. Explain why.